

Type Checker Overview

August 28, 2006

By

David Wallwork – Software Architect, BASIS International Ltd.

This paper provides an overview of the BBJ type check system introduced in BBJ 6.0 and discusses how that type check system interacts with other BBJ features.

Contents

Introduction

Static Type Checking vs. Dynamic Type Checking

Assignment Compatibility

BBJ Native Types and Assignment Compatibility

Transitivity, Autoboxing, Widening of Java Types and Assignment to Object

Assignment Compatibility - A More Formal Definition

Errors Generated by the Type Check System

Using bbjcp1 to do Static Type Checking

Type Check Errors in Existing Code

Conclusion

Type Checker Overview

Introduction

BBj now includes a new type check system that forms the foundation for the following features that BASIS introduced in BBj 6.0:

- Custom objects
- Code completion in the IDE
- **declare** verb
- Type checking options of bbjcp1

The developer who has a basic understanding of object-oriented concepts can use custom objects and code completion without concern about the details of the type check system. In order to take full advantage of the **declare** verb and the type checking options of bbjcp1, the developer needs a clear understanding of the underlying type check system.

This paper discusses the type check system and how it can help developers discover their programming errors during development rather than at runtime and how to write more stable and robust code.

Static Type Checking vs. Dynamic Type Checking

Various languages employ different approaches to type checking. One way to categorize these approaches is to group them as either *static* or *dynamic* type checking. Static implies that the type checking is performed at compile time while dynamic implies that the type checking is done at runtime. Traditionally, the BBx language provides static type checking for the interactions between strings and numbers. For example, a syntax error results in BBx when attempting to add a string to a number or when passing a string to a BBj function that requires a number. These types of errors occur when using bbjcp1 or when loading a program, and are examples of static type checking.

BBj also performs dynamic type checking. Consider the fact that the developer can create expressions that evaluate to strings or to numbers or (using embedded Java) to arbitrary Java types. The BBj compiler allows assignment from embedded Java expressions both to numeric variables and to string variables but will throw an error at runtime if the type is incorrect. For example, the parser accepts the statement `A = new java.lang.String("abc")` and does not marked it as a syntax error. However, at runtime the statement will result in a runtime error because a string expression cannot be assigned to a number variable. The error generated at runtime is an example of dynamic type checking.

There are benefits both to dynamic type checking and static type checking.

- + **Dynamic type checking** allows flexibility and often reduces the number of lines of code.
- + **Static type checking** allows programmers to eliminate many runtime errors by finding them during compilation.

BBj 6.0 introduces a static type checker to allow developers to take full advantage of static type checking to assure that their programs do not contain hidden problems that will result in runtime errors. Developers will immediately recognize the advantage of having errors reported during a type check operation rather than during runtime. Another advantage of static type checking that is not as apparent is that static type checking provides complete coverage of a program. With dynamic type checking, a statement is not checked until it is actually executed. By contrast, with static type checking all statements are checked during the type check operation.

Consider the following program that contains an error:

```
bbj! = bbjapi ()
if x = 5
    sg! = bbj!.GETSysGUI () ;      rem this is an error. there is no such
function
endif
```

Developers can test and deploy this code without having to execute the line that contains an error. The error will only be discovered when the code is executed with a value of 5 for the variable x.

The next code sample illustrates how to modify this code to use declared variable types:

```
declare BBjAPI bbj!
declare BBjSysGui sysGui!

bbj! = bbjapi ()
if x = 5
    sysGui! = bbj!.getSysGui () ; rem error there is no such function
endif
```

The BBj type check system can recognize the error in this modified code because the type of the variable `bbj!` has been declared to be **BBjAPI**. In addition, the type check system can determine that **BBjAPI** does not have a method `getSysGui ()`. If all variables in a program have been declared, then the static type check operation is able to find many hidden program errors that would otherwise have been found only at runtime.

Before describing the type check system more fully, we need to discuss assignment compatibility. The next several sections provide information about assignment compatibility in general and how it is applied in BBj. A precise description of the static type checking capabilities of the type check system appears in the section ***Static Type Checking Using bbjcpl Options***.

Assignment Compatibility

In BBj, all variables and all expressions have an associated type. We are concerned with whether a variable or an expression of one type can be assigned to a variable of another (possibly different) type or can be used in place of a variable of another (possibly different) type. For example, an application might be working with the types *Animal*, *Dog*, and *Tree* where the type *Dog* is derived from the type *Animal*. Intuitively, it makes sense that an expression of type *Dog*

could be assigned to a variable of type *Dog* or to a variable of type *Animal*, but not to a variable of type *Tree*. Similarly, a variable of type *Dog* could be passed as an argument to a function that expected a variable of type *Dog* or that expected a variable of type *Animal*, but could not be passed to a function that expected a variable of type *Tree*.

In its simplest form, assignment compatibility says that a type *A* is “assignment compatible¹” to a type *B* if *A* and *B* are the same type or if *A* is a subtype of *B*. We will reach a more concise definition of assignment compatibility in the following sections.

BBj Native Types and Assignment Compatibility

In BBj 6.0, there are three “primitive types,” each corresponding to one of the original variable types in the BBx language: **BBjNumber**, for numeric expressions; **BBjString** for string expressions; and **BBjInt** for integer expressions. In order to allow the seamless integration of embedded Java in a BBj application, BBj applies specific rules about assignment compatibility to these primitive types.

The Java types `java.lang.String` and `byte[]` are assignment compatible to **BBjString**. This means that a Java expression that evaluates to `java.lang.String` or to `byte[]` may be assigned to a traditional BBj string variable **A\$**. Similarly, a **BBjString** is assignment compatible to `java.lang.String` and is assignment compatible of `byte[]` which means that a traditional BBj string variable **A\$** may be used in a Java method that accepts a `java.lang.String` or a `byte[]`.

A **BBjNumber** is assignment compatible to any Java primitive numeric type (`int`, `float`, `double` etc) and to `boolean` and to the corresponding Java wrapper types such as `Integer`, `Float`, and `Boolean`. The Java primitive numeric types, `boolean` and the corresponding wrappers, are all assignment compatible to **BBjNumber**. In other words, the BBj programmer can use a traditional BBj numeric variable **x** in any method that requires one of these types and can assign an expression that returns one of these types to a traditional BBj numeric variable.

A **BBjInt** is assignment compatible to Java `int`, `Integer`, `boolean`, and `Boolean`, and these types are assignment compatible to **BBjInt**. This means that a traditional BBj integer variable **x%** can be used in any method that requires one of these Java types and that an expression that evaluates to one of these types can be assigned to a variable **A%**.

All of which means that the BBj primitive types (**A**, **A\$**, **A%**) can be used to interact with embedded Java in the way in which a developer would expect.

¹ King, Stuart. *Irie Pascal Programmer's Reference Manual*. <http://www.iriertools.com/iriepascal/progref364.html>
“Assignment compatibility is a relationship between values and types, and for any given value and type the relationship either exists between them or it does not (i.e. the value is either assignment compatible with the type or it is not).”

Transitivity, Autoboxing, Widening of Java Types and Assignment to Object

The first time reader may choose to skip this section. However, this section contains valuable details that provide a precise explanation of assignment compatibility in BBj.

Transitivity

Assignment compatibility is transitive. This means that if a type **A** is assignment compatible to a type **B** and the type **B** is assignment compatible to the type **C**, then the type **A** is assignment compatible to the type **C**. Extending our earlier example, we might have an application that had the types **Animal**, **Dog**, **Collie**, and **Tree** where **Collie** is derived from the type **Dog**. Since a **Collie** is assignment compatible to a **Dog** and a **Dog** is assignment compatible to an **Animal**, a **Collie** is assignment compatible to an **Animal** and a variable of type **Collie** can be used wherever a variable of type **Animal** is required.

Autoboxing

BBj will convert between the Java primitive types and their corresponding wrappers (**int** and **Integer**, **long**, and **Long**, etc.) as needed. In other words, in BBj each Java primitive type is assignment compatible to its corresponding wrapper type and each wrapper type is assignment compatible to its corresponding primitive. This is similar to the autoboxing that Sun introduced in Java version 1.5.

Widening of Java Types

BBj will widen Java primitive types. In BBj, **byte** is assignment compatible to **short** is assignment compatible to **int** is assignment compatible to **long** is assignment compatible to **float** is assignment compatible to **double**.

Assignment to Object

In BBj, any value including a Java primitive type is assignment compatible to the type of `java.lang.Object`.

Assignment Compatibility - A More Formal Definition

A type **A** is assignment compatible to a type **B** if any of the following conditions are met:

- **B** is a Java interface and **A** implements **B**
- **B** is a Java class and **A** extends **B**
- **B** is a BBj custom object interface and **A** implements **B**
- **B** is a BBj custom object class and **A** extends **B**
- **A** is assignment compatible to **B** for one of the reasons discussed in the section ***BBj Native Types and Assignment Compatibility***
- **A** is assignment compatible to **B** for one of the reasons discussed in the section ***Transitivity, Autoboxing, Widening of Java types and Assignment to Object***

Declare Verb and Cast Function

BBj 6.0 introduced the `declare` verb and the `cast` function which play an important role in the type checking system.

The `declare` verb allows the programmer to associate a specific type to an object variable. The statement `declare java.util.HashMap map!` associates the type `java.util.HashMap` to the variable `map!`.

The `cast` function instructs BBj to associate a specific type to an expression. The expression `cast(java.lang.String, map!.get("key"))` causes BBj to associate the type `java.lang.String` with the return value of the expression `map!.get("key")`.

Declared Values and Undeclared Values

As stated earlier, all variables and expressions have an implicit type in BBj. In the case of a variable `A!`, the implicit type is `java.lang.Object`. In the case of `A`, `A$`, and `A%`, the implicit type is `BBjNumber`, `BBjString`, and `BBjInt`. In addition to this implicit type, a value (a variable or an expression) may have a declared type. When a value has a declared type, it is said to be *declared*. A value that is not declared is said to be *undeclared*.

The key to understanding the type checking system is in understanding which values (variables and expressions) are declared.

- Variables referred to by a `declare` statement are declared
- The return value of the `cast` function is declared
- Custom object fields are declared
- The return value of a custom object method is declared
- The parameters of a custom object method are declared
- The return value of a method called on a declared value is declared
- The return value of a static java method is declared*
- A reference to a Java static field is declared*
- The return value of the `new` verb is declared*
- The return value of the function `BBjAPI()` * is declared
- A numeric expression is declared if any operand of the expression is declared

*see discussion of disabling type check system in subsequent section

Errors Generated by the Type Check System

We are now in a position to describe the rules that the BBj type check system enforces. Any statement that violates one of these rules will cause an error at runtime.

A statement contains a type check error if it attempts to do any of the following

- Assign an undeclared variable to a declared variable
- Assign an undeclared expression to a declared variable
- Execute a numeric expression in which some operands are declared and others are not

- Invoke a non-existent method on a declared variable or on the resultant of a declared expression
- Assign a declared variable or a declared expression to another declared variable such as the left hand side (**LHS**) variable if the type of the declared variable or expression is not assignment compatible to the declared type of the variable to which it is being assigned (the **LHS** variable)
- Use a declared variable or a declared expression as a parameter to a method call if the type of the declared variable or expression is not assignment compatible to the type of the parameter
- Use the **cast** function to cast a declared variable or a declared expression to a type if the type of the variable or expression is not assignment compatible to the type to which it is being cast

Using bbjcp1 to do Static Type Checking

If a program has declared all variables, then BBj is able to do static type checking to examine the entire program and find all statements that will generate a type check error at runtime. If a program contains undeclared variables, then BBj cannot always determine whether a statement will generate a type check error at runtime.

BBj 2.0 introduced four additional command line options for bbjcp1: **-t**, **-w**, **-p**, and **-c**. The three options **-w**, **-p**, and **-c** can only be used in conjunction with the **-t** option.

Using the **-t** option enables static type checking. After bbjcp1 compiles the given programs to output files, it will check the output files and generate a listing of all program lines contain type check errors.

Using the **-w** option enables warnings about undeclared code. After compiling the given programs and generating a list of all programs containing type check errors, bbjcp1 will also generate a listing of all lines that it could not check because they contained undeclared variables.

Programs that use custom objects will often contain **use** statements that tell the program where to find the definitions of custom objects defined in other files. At runtime, BBj applies the standard prefix algorithm to the filenames in these **use** statements in order to resolve the names to files on disk. When using bbjcp1 to type check a program, it does not have access to the prefix that will be in affect when the programs are actually run. Actually, bbjcp1 provides two options to provide a prefix for the static type checker to use when searching for class definitions.

Using the **-p<directory list>** option tells the static type checker to search for files containing custom object class definitions in the directories specified. Enter the directory names separated with semicolons on Windows environments or by colons on UNIX-like environments, similar to the **PATH** environment variable on the respective operating systems.

Using the **-c<configFile>** option specifies a configuration file that the static type checker can use. The static type checker will use the prefix list contained in the **<configFile>** when

resolving the filenames contained in `use` statements. Since `bbjcpl` does not have access to the BBJ filesystem, it will ignore entries that specify data server syntax.

It is highly recommended that developers check all code using `bbjcpl -t -w` in order to discover all type check problems prior to deploying their code.

When type checking existing code that makes heavy use of objects, this may print out a large number of warnings that adding `declare` statement to the code will address. However, adding a single `declare` statement often eliminates many warning statements so this process is not as daunting as it may first appear. Once the process has been completed, the developer can trust that the code is free from type check errors that otherwise might become runtime problems.

Type Check Errors in Existing Code

There is a very small set of statements that runs correctly in previous versions of BBJ but that will generate a type check error in BBJ 6.0. This section describes these statements and the options available to a developer to run such statements in BBJ 6.0.

First, consider the following code that runs in previous versions of BBJ but will not run in BBJ 6.0:

```
0010 open (unt) "X0"

0020 x = 0
0030 sysGui! = bbjapi().getSysGui()
0040 win! = sysGui!.addWindow(10,10,600,400,"test")
0050 if x = 0
0060     win!.addTabCtrl(200,20,20,200,100)
0070 else
0080     win!.addButton(200,20,20,200,100)
0090 endif
0100 bbjapi().getSysGui().getWindow(0).getControl(200).setNumTabs(3)
```

By referring to the section ***Declared Values and Undeclared Values***, we see that the return value of `bbjapi()` is declared. If we look closely at the expression in line 100 we see the following:

```
bbjapi() is declared and has type BBJAPI
bbjapi().getSysGui() is declared and has type BBJSysGui
bbjapi().getSysGui().getWindow(0) is declared and has type BBJWindow
bbjapi().getSysGui().getWindow().getControl(200) is declared and has
type BBJControl
```

BBJ 6.0 will generate a type check error on line 100 because **BBJControl** does not have a method `setNumTabs()`.

Earlier BBJ versions evaluated a `bbjapi().getSysGui().getWindow().getControl(200)` expression and the value happened to be a **BBJTabCtrl**. After it is evaluated, BBJ invokes the method `setNumTabs()` on the resultant **BBJTabCtrl**. But BBJ 6.0 examines the return type of

the expression and discovers that the declared return type of `getControl()` is a **BBjControl**. BBj also realizes that **BBjControl** does not have a method `setNumTabs()` and so it throws a type check error.

At first this might seem to be undesirable behavior. However, consider what would happen if the code on line 20 were modified to set `x` to some other value. The code succeeds if `x` happens to have been set to zero but fails if it is set to a different value. In BBj 6.0, the code always fails but a developer can easily modify it to succeed. It is far better to have code that will always fail and be checked with a static type checker than to have code that will sometimes fail and cannot be checked.

It is possible to contrive code that will also run in previous versions but not in BBj 6.0. But it would require using functionality that is new to BBj 6.0 (custom objects and/or BBjEvents) or creating custom Java code that was especially structured to have the problem.

Granted, this entire issue may have already received more attention than it merits. However, for the interested reader, we now provide three ways in which this problem can be resolved if you find that it occurs in your existing code.

If you find code that runs in previous versions of BBj but not in BBj 6.0, do one of the following:

- Restructure code to use temporary variable
- Restructure code to use cast function
- Disable the type check system

Restructuring Code to Use Temporary Variable

If line 100 is modified to use a temporary variable, it might look like this:

```
0100 x! = bbjapi().getSysGui().getWindow(0).getControl(200)
0101 x!.setNumTabs(3)
```

In this new code, `x!` is not declared and so the call to `x!.setNumTabs()` will not fail.

Restructuring Code to Use Cast Function

If line 100 is modified to use the cast function, it might look like this:

```
0100 cast(BBjTabCtrl, bbjapi().getSysGui().getWindow(0).getControl(200)).setNumTabs(3)
```

In this new code, the cast expression has a return type of **BBjTabCtrl** and so it is legal to call `setNumTabs()` on the cast expression.

Disabling the Type Check System

To disable the type check system, add a **STBL** entry having the key of `"!COMPAT"` and a value `"LEGACY_TYPECHECK=ON"` in the configuration file or add it programmatically using the following statement:

```
dummy$ = STBL("!COMPAT", "LEGACY_TYPECHECK=ON")
```

Disabling the type check system will result in the following:

- The return value of a static Java method is no longer a declared value
- A reference to a Java static field is no longer a declared value
- The return value of the **new** verb is no longer a declared value
- The return value of the function **BBjAPI ()** is no longer a declared value

When the type check system is disabled, the code runs without modification.

Conclusion

The new command line options on bbjcpl provide the developer access to the static type checking capabilities of BBj. By using static type checking, developers can discover and eliminate many coding errors that they could only find previously when the code failed at runtime.